Database Connectivity and Management

Introduction:
Database connectivity is an essential aspect of modern software development. Databases store and manage vast amounts of structured data, making them crucial for many applications. In this study material, we will explore the fundamentals of database connectivity and management, covering topics such as ODBC API, JDBC API, querying databases, and working with popular database systems like MySQL and Oracle9i. Additionally, we will delve into creating and processing HTML forms, which often serve as interfaces to interact with databases.

1. Database Management:

Databases are organized collections of data used to store, manage, and retrieve information.
Database management involves creating, maintaining, and optimizing databases to ensure data integrity and availability.

2. ODBC API and JDBC API:

> ODBC (Open Database Connectivity):

ODBC is a standard API for connecting and accessing relational databases.
It enables communication between applications and various database systems.
ODBC provides a consistent interface for different databases.

> JDBC(Java Database Connectivity)

Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access any kind of tabular data, especially relational databases. It is part of the Java Standard Edition platform, from Oracle Corporation. It acts as a middle layer interface between java applications and databases.

The JDBC classes are contained in the Java Package java.sql and javax.sql.
JDBC helps you to write Java applications that manage these three programming activities:

Connect to a data source, like a database.
Send queries and update statements to the database

Retrieve and process the results received from the database in answer to your query
Structure of JDBC

JDBC drivers are client-side adapters (installed on the client machine, not on the server) that convert requests from Java programs to a protocol that the DBMS can understand. There are 4 types of JDBC drivers:

Type-1 JDBC-ODBC bridge driver
Type-2 Native-API driver
Type-3 Network Protocol driver
Type-4 Thin driver

Type-1 driver

Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. Type-1 driver is also called Universal driver because it can be used to connect to any of the databases.

As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secure.
The ODBC bridge driver is needed to be installed in individual client machines.
Type-1 driver isn't written in java, that's why it isn't a portable driver.
This driver software is built-in with JDK so no need to install separately.
It is a database independent driver.

Type-2 driver

The Native API driver uses the client -side libraries of the database. This driver converts JDBC method calls into native calls of the database API. In order to interact with different databases, this driver needs their local API, that's why data transfer is much more secure as compared to type-1 drivers.

Driver needs to be installed separately in individual client machines
The Vendor client library needs to be installed on the client machine.
Type-2 driver isn't written in java, that's why it isn't a portable driver
It is a database dependent driver.

Type-3 driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. Here all the database connectivity drivers are present in a single server, hence no need for individual client-side installation.

Type-3 drivers are fully written in Java, hence they are portable drivers.
No client side library is required because of the application server that can perform many tasks like auditing, load balancing, logging etc.
Network support is required on client machines.
Maintenance of Network Protocol drivers becomes costly because it requires database-specific coding to be done in the middle tier.
Switch facility to switch over from one database to another database.

Type-4 driver

Type-4 drivers are also called native protocol drivers. This driver interacts directly with the database. It does not require any native database library, that is why it is also known as Thin Driver.

Does not require any native library and Middleware server, so no client-side or server-side installation.
It is fully written in Java language, hence they are portable drivers.

Which Driver to use When?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is type-4.
If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only

JDBC components

The JDBC core comes with the following interfaces and classes:

- Driver: This is the interface that controls communication with the database server. It also withdraws information associated with driver objects.
- Driver Manager: It manages any required set of JDBC drivers
- Connection: This is an interface or session that houses all the methods to connect to any database.
- Statements: This is used to carry out a static SQL statement
- ResultSet: This is used to access the result row-by-row

Relationship between SQL (a relational database) and JDBC

A relational database like SQL is a structured repository that stores records in tables with columns and rows. Although NoSQL has gained popularity over the past decade, relational databases like SQL are still the most commonly used type of datastore. The main language used by data architects to perform various tasks related to a relational database is SQL. It can perform various tasks such as creating, reading, and updating records.

In Java JDBC connection, the JDBC acts as an adapter layer that provides adaptability to SQL from Java. This enables the Java developers to connect to a database and allows them to perform various tasks like managing responses and queries.

How to create a Java JDBC connection

We have learned what a JDBC is, what it is used for, its components, driver types, and its relationship with relational databases like SQL. Now, we will be going through detailed steps through which we use JDBC to create a connection to a database in Java. Here is the 7 step process to create a Java JDBC connection:

1. Import the packages:

This includes uploading all the packages containing the JDBC classes, interfaces, and subclasses used during the database programming. More often than not, using import **java.sql.*** is enough. However, other classes can be imported if needed in the program.

2. Load/Register the drivers:

Before connecting to the database, we'll need to load or register the drivers once per database. This is done to create a communication channel with the database. Loading a driver can be done in two ways:

For MySQL: Class.forName("com.mysql.jdbc.Driver");

For Oracle: Class.forName("oracle.jdbc.driver.OracleDriver");

3. Establish a connection:

For the next step here, the getConnection() method is used to create a connection object that will correspond to a physical connection with the database. To get the getConnection() to access the database, the three parameters are a username, string data type URL, and a password. Two methods can be used to achieve this:

- getConnection(URL, username, password): This uses three parameters URL, a password, and a username
- getConnection(URL): This has only one parameter - URL. The URL has both a username and password. There are several JDBC connection strings for different relational databases and some are listed below: a. IBM DB2 database: jdbc:db2://HOSTNAME:PORT/DATABASE_NAME b. Oracle database: jdbc:oracle:thin:@HOST_NAME:PORT:SERVICE_NAME c. My SQL database: jdbc:mysql://HOST_NAME:PORT/DATABASE_NAME

```
// Database URL, username, and password
String url = "jdbc:mysql://localhost:3306/Employee"; // MySQL
//String url1 = "jdbc:oracle:thin://@localhost:1521/Student"; //Oracle
String username = "root";
String password = "Root@123";
// Initialize the connection
try {
        Connection connection = DriverManager.getConnection(url, username,
password);

        // Check if the connection is successful
        if (connection != null) {
```

```
                System.out.println("Connected to the database!");
            } else {
                System.out.println("Failed to connect to the database!");
        }
} catch (SQLException e) {
        // Handle any exceptions that may occur during the connection process
        e.printStackTrace();
}
```

4. Create a statement:

The statement can now be created to perform the SQL query when the connection has been established. There are three statements from the createStatement method of the connection class to establish the query. These statements are

- **Statement**: This is used to create simple SQL statements with no parameter. An example is: Statement statement1 = conn.createStatement();. This statement returns the ResultSet object.
- **PreparedStatement**: This extends the Statement interface. It improves the application's performance because it has more features and compiles the query only once. It is used for precompiled SQL statements that have parameters.

```
PreparedStatement ps = connection.prepareStatement("INSERT INTO emp VALUES(?,?,?,?)");
//Parameterized Query
ps.setInt(1,6); // Employee ID
ps.setString(2,"Ravi"); // Employee Name
ps.setString(3,"Team Lead"); // Designation
ps.setInt(4,1020);
System.out.println("Prepared Procedure success :" + ps.executeUpdate());
```

- **CallableStatement**: CallableStatements also extends the PreparedStatement interface. It is used for SQL statements with parameters that invoke procedure or function in the database. It is simply created by calling the prepare all method of the connection object.

```
CallableStatement t = connection.prepareCall("{call get_emp(?)}");
// get_emp is Stored Procedure
t.setInt(1,6);
int mn = t.executeUpdate();
System.out.println("Store Procedure success :" + mn);
```

5. Execute the query:

This uses a type statement object to build and submit SQL statements to a database. It has four distinct methods:

- ResultSet executeQuery(String sql)
- int executeUpdate(String sql)
- boolean execute(String sql)
- int[] executeBatch()

ResultSet is obtained by calling the executeQuery method on the Statement instance. Initially, the cursor of ResultSet points to the position before the first row. The method next of ResultSet moves the cursor to the next row. It returns true if there is a further row otherwise it returns false.

In Java, the methods execute(), executeUpdate(), executeQuery(), and executeBatch() are part of the java.sql.Statement interface, and they are used to execute SQL statements in the context of a database connection. Each of these methods serves a different purpose:

The execute() method is a versatile method that can execute any SQL statement, including queries, updates, and stored procedures.

It returns a boolean value (true if the result is a ResultSet, false if the result is an update count or there is no result).

Example:

```
boolean hasResultSet = statement.execute("SELECT * FROM my_table");

if (hasResultSet) {

    ResultSet resultSet = statement.getResultSet();

    // Process the result set

} else {

    int updateCount = statement.getUpdateCount();

    // Process the update count

}
```

> **The executeUpdate()** method is used to execute SQL statements that modify the database, such as INSERT, UPDATE, DELETE, or DDL (Data Definition Language) statements.

It returns an integer that represents the number of rows affected by the SQL statement.

Example: int rowsAffected = statement.executeUpdate("UPDATE my_table SET column1 = 'new_value' WHERE condition");

> **The executeQuery()** method is specifically designed for executing SQL SELECT statements, which retrieve data from the database.

It returns a ResultSet object that represents the result set of the query.

Example:ResultSet resultSet = statement.executeQuery("SELECT * FROM my_table");

```
while (resultSet.next()) {

    // Process the rows of the result set

}
```

> **The executeBatch()** method is used for batch processing of SQL statements. It allows you to group multiple SQL statements and execute them together as a batch.

It returns an array of integers, each representing the update count for a statement in the batch.

Example:

```
statement.addBatch("INSERT INTO my_table (column1, column2) VALUES ('value1', 'value2')");

statement.addBatch("UPDATE my_table SET column1 = 'new_value' WHERE condition");

int[] updateCounts = statement.executeBatch();
```

Key Differences:

execute() is a general-purpose method that can execute any SQL statement and returns a boolean value.

executeUpdate() is used for executing SQL statements that modify the database and returns the number of affected rows.

executeQuery() is specifically for executing SELECT statements and returns a ResultSet containing query results.

executeBatch() is used for batch processing of multiple SQL statements and returns an array of update counts.

It's important to choose the appropriate method based on the type of SQL statement you are executing, as using the correct method ensures that you handle the result or update count correctly.

6. Retrieve results:

When queries are executed using the executeQuery() method, it produces results stored in the ResultSet object. The ResultSet object is then used to access the retrieved data from the database.

JDBC Scrollable ResultSet Example

Whenever we create an object of ResultSet by default, it allows us to retrieve in the forward direction only and we cannot perform any modifications on ResultSet object. Therefore, by default, the ResultSet object is non-scrollable and non-updatable ResultSet.

In this tutorials, I am going to tell you how to make a ResultSet object as scrollable.

JDBC Scrollable ResultSet :

A scrollable ResultSet is one which allows us to retrieve the data in forward direction as well as backward direction but no updations are allowed. In order

to make the non-scrollable ResultSet as scrollable ResultSet we must use the following *createStatement()* method which is present in Connection interface.

```
public Statement createStatement(int Type, int Mode);
```

Here t*ype* represents the type of scrollability and *mode* represents either read only or updatable. The
value of Type and the Modes are present in ResultSet interface as constant data members and they are:

- TYPE_FORWARD_ONLY -> 1
- TYPE_SCROLL_INSENSITIVE -> 2
- TYPE_SCROLL_SENSITIVE -> 3

We can pass the above constants to ResultSet as below:

```
Statement st=con.createStatement (
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY );
```

```
ResultSet rs=st.executeQuery ("select * from empleyee");
```

ResultSet Concurrency Values

A `Statement` can return result sets which are read-only or updatable, specified by one of the following constants defined in the `ResultSet` interface:

- **`CONCUR_READ_ONLY`**: the result set cannot be used to update the database (default).

- **`CONCUR_UPDATABLE`**: the result set can be used to update the database.

For example, if you want to scroll through the result set but don't want to update its data, create `Statement` a like this:

```
1 Statement statement = connection.createStatement(
2 ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Whenever we create a ResultSet object, by default, constant **TYPE_FORWARD_ONLY** as a Type and **CONCUR_READ_ONLY** as mode will be assigned.

ResultSet interface provides us several methods to make an ResultSet as Scrollable ResultSet below is the list of methods available in ResultSet interface.

- public boolean next (); It returns true when rs contains next record otherwise false.
- public void beforeFirst (); It is used for making the ResultSet object to point to just before the first record (it is by default)
- public boolean isFirst (); It returns true when rs is pointing to first record otherwise false.
- public void first (); It is used to point the ResultSet object to first record.
- public boolean isBeforeFirst (); It returns true when rs pointing to before first record otherwise false.
- public boolean previous (); It returns true when rs contains previous record otherwise false.
- public void afterLast (); It is used for making the ResultSet object to point to just after the last record.
- public boolean isLast (); It returns true when rs is pointing to last record otherwise false.
- public void last (); It is used to point the ResultSet object to last record.
- public boolean isAfterLast (); It returns true when rs is pointing after last record otherwise false.
- public void absolute (int); It is used for moving the ResultSet object to a particular record either in forward direction or in backward direction with respect to first record and last record respectively. If int value is positive, rs move in forward direction to that with respect to first record. If int value is negative, rs move in backward direction to that with respect to last record.
- public void relative (int); It is used for moving rs to that record either in forward direction or in backward direction with respect to current record.

JDBC Scrollable ResultSet Example :

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;


public class ScrollResultSet {

    public static void main(String[] args) throws Exception {

        Class.forName("com.mysql.jdbc.Driver");

        Connection con = DriverManager.getConnection(

                "jdbc:mysql://localhost:3306/jgmca", "root",

                "123456");

        Statement st =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,

                ResultSet.CONCUR_READ_ONLY);

        ResultSet rs = st.executeQuery("select * from student");

        System.out.println("RECORDS IN THE TABLE...");

        while (rs.next()) {

            System.out.println(rs.getInt(1) + " -> " +
rs.getString(2));

        }

        rs.first();

        System.out.println("FIRST RECORD...");
```

```
        System.out.println(rs.getInt(1) + " -> " +
rs.getString(2));

        rs.absolute(3);

        System.out.println("THIRD RECORD...");

        System.out.println(rs.getInt(1) + " -> " +
rs.getString(2));

        rs.last();

        System.out.println("LAST RECORD...");

        System.out.println(rs.getInt(1) + " -> " +
rs.getString(2));

        rs.previous();

        rs.relative(-1);

        System.out.println("LAST TO FIRST RECORD...");

        System.out.println(rs.getInt(1) + " -> " +
rs.getString(2));

        con.close();

    }

}
```

7. Close the connections:

The JDBC connection can now be closed after all is done. The resource has to be closed to avoid running out of connections. It can be done automatically using ‘conn.close();’. But for versions of Java 7 and above, it can be closed using a try-catch block

Important Notes:

1) DriverManager is class
2) Connection, Statement and ResultSet are interfaces.
3) commit, rollback are methods of Connection Interface.
4) getConnection is a static method of the DriverManager class.
5) JDBC-ODBC bridge work with Multiple-Thread
6) JDBC ODBC classes and interfaces are part of java.sql and javax.sql packages.

```
// This code is for establishing connection with MySQL
// database and retrieving data
// from db Java Database connectivity

/*
*1. import --->java.sql
*2. load and register the driver ---> com.jdbc.
*3. create connection
*4. create a statement
*5. execute the query
*6. process the results
*7. close
*/

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

class JDBCConnectivityExample {
public static void main(String[] args) {
// Database URL, username, and password
String url = "jdbc:mysql://localhost:3306/Employee";
```

```
String username = "root";
String password = "Root@123";

// Initialize the connection
try {

Connection connection = DriverManager.getConnection(url, username, password);

// Check if the connection is successful
if (connection != null) {
System.out.println("Connected to the database!");
// You can now use 'connection' to execute SQL queries
Scanner scanner = new Scanner(System.in);
for(int i=0;i<2;i++) {
System.out.println("Enter ENo: ");
int no = scanner.nextInt();

System.out.println("Enter EName: ");
String name = scanner.next();

System.out.println("Enter EDesignation: ");
String designation = scanner.next();

PreparedStatement ps = connection.prepareStatement("INSERT INTO emp VALUES(?,?,?)");
ps.setInt(1,no);
ps.setString(2,name);
ps.setString(3,designation);
System.out.println("Prepared Procedure success :" + ps.executeUpdate());
}

scanner.close();


ResultSet resultSet = connection.createStatement().executeQuery("SELECT * FROM emp");

// Process the query result (e.g., print data)
while (resultSet.next()) {
// Retrieve data from the result set
int id = resultSet.getInt("empid");
```

```
String ename = resultSet.getString("empname");
String desn = resultSet.getString("designation");

// ... retrieve other columns as needed

// Process or print the data
System.out.println("ID: " + id + ", FirstName: " + ename + ", LastName: " + desn);
}
} else {
System.out.println("Failed to connect to the database!");
}

// Close the connection when done
connection.close();
} catch (SQLException e) {
// Handle any exceptions that may occur during the connection process
e.printStackTrace();
}
}
}
```

Unit – 2 Introduction To Java Servlets

Java Servlets are a fundamental technology for developing dynamic web applications. They are server-side components that extend the capabilities of a web server, enabling the processing of client requests and generating dynamic responses. This study material introduces you to Java Servlets, their role in web development, and comparisons with other web technologies.

Characteristics of Servlets

Servlets can be used to develop a variety of web-based applications. As Servlets are written using Java, they can use the extensive power of the Java API, such as networking and URL access, multithreading, database connectivity, internationalization, remote method invocation (RMI), and object serialization. The characteristics of Servlets that have gained them widespread acceptance are as follows:

- **Servlets are efficient:** The initialization code for a Servlet is executed only when the Servlet is executed for the first time. Subsequently, the Servlet's requests are processed by its service() method. This helps increase the efficiency of the server by avoiding the creation of unnecessary processes.
- **Servlets are robust:** Servlets are based on Java; they provide all the powerful features of Java, such as exception handling and garbage collection, which make them robust.
- **Servlets are portable:** Servlets are also portable because they are developed in Java. This enables easy portability across the web servers.
- **Servlets are persistent:** Servlets increase the system's performance by preventing frequent disk access. For example, if a customer logs on to www.EarnestOnline.com, the customer can perform many activities, such as checking for the balance, applying for a loan, and so on. In every stage, the customer needs to be authenticated by checking for the account number against

the database; instead of checking for the account number against the database every time, Servlets retain the account number in the memory till the user logs out of the website.

1. Webserver & Introduction to Servlets:

A web server is a software application responsible for serving web content to clients (browsers).
Java Servlets are server-side Java components that process requests, execute logic, and generate dynamic web content.
Servlets are hosted within a web server and are used to create web applications.

2. Comparison between Servlets and Applets:

Servlets:

- Run on the server-side.
- Used for creating dynamic web applications.
- Typically respond to HTTP requests.
- Servlet does not require the browser to be java enabled.

Applets(Deprecated Technology):

- Run on the client-side.
- Used for creating interactive web content.
- Embedded in web pages and executed in the user's browser.
- Applet required the browser to be java enabled because they execute on the webserver

3. Comparison between Servlets and other server-side technologies:

- **Servlets vs. CGI (Common Gateway Interface):**

  Servlets are more efficient than CGI because they run in the same address space as the web server.

  Servlets are written in Java, providing platform independence.

Difference between Servlet and CGI

Servlet	CGI(Common Gateway Interface)
Servlets are portable and efficient.	CGI is not portable
In Servlets, sharing data is possible.	In CGI, sharing data is not possible.
Servlets can directly communicate with the webserver.	CGI cannot directly communicate with the webserver.
Servlets are less expensive than CGI.	CGI is more expensive than Servlets.
Servlets can handle the cookies.	CGI cannot handle the cookies.

- **Servlets vs. JSP (JavaServer Pages):**

  Servlets are Java classes that define request handling logic.

  JSP is a technology that allows embedding Java code in HTML to create dynamic web pages.

Active Server Pages

- ASP is a server-side scripting language that has been developed by Microsoft.
- ASP enables a developer to combine HTML and a Scripting language on the same web page.
- JavaScript and VBScript are two scripting languages that are supported by ASP.
- The limitation of ASP is that it is not compatible with all the web servers.
- The other web servers need specific plug-ins to be installed to support ASP. However, adding a plug-in can decrease the performance of the system.

GET & POST Method

In Java servlets, the GET and POST methods are two of the most common HTTP request methods used to interact with web applications. These methods are used to retrieve and send data to and from the server. Below, I'll explain the differences and typical use cases for the GET and POST methods in Java servlets:

1. GET Method:

The GET method is used to request data from the server. It appends data to the URL as query parameters.
The data is visible in the URL, making it suitable for non-sensitive data.

GET requests are idempotent, meaning they can be repeated without changing the server's state.
GET requests are cached by browsers, making them suitable for retrieving resources like images, stylesheets, or JavaScript files.
In servlets, you can retrieve GET parameters using the request.getParameter() method.
Example of a GET request in a Java servlet:

Example
```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
   String parameterValue = request.getParameter("parameterName");
   // Process the parameterValue and generate a response
}
```
2. POST Method:

The POST method is used to send data to the server in the request body, rather than in the URL.
POST requests are suitable for sending sensitive or large amounts of data.
They are not cached by browsers and are not idempotent, as they can change the server's state with each request.
In servlets, you can retrieve POST parameters using the request.getParameter() method.
Example of a POST request in a Java servlet:

Example
```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
   String parameterValue = request.getParameter("parameterName");
   // Process the parameterValue and generate a response
}
```
In a typical web application, you may use the GET method to retrieve resources or perform read-only operations. For example, when viewing a web page or searching for information.

The POST method is used when you want to send data to the server to perform actions that modify server state. For example, when submitting a form that updates data in a database.

It's important to choose the appropriate method based on the specific requirements of your web application. In practice, web applications often use a combination of both GET and POST methods to handle various types of requests

Difference Between dopost and doget in Servlet

Http protocol mostly use either get or post methods to transfer the request. post method are generally used whenever you want to transfer secure data like password, bank account etc.

	Get method	Post method
1	Get Request sends the request parameter as query string appended at the end of the request.	Post request send the request parameters as part of the http request body.
2	Get method is visible to every one (It will be displayed in the address bar of browser).	Post method variables are not displayed in the URL.
3	Restriction on form data, only ASCII characters allowed.	No Restriction on form data, Binary data is also allowed.
4	Get methods have maximum size is 2000 character.	Post methods have maximum size is 8 mb.
5	Restriction on form length, So URL length is restricted	No restriction on form data.
6	Remain in browser history.	Never remain the browser history.

Javax.Servlet Package

The javax.servlet package is a core part of the Java Servlet API, which provides a framework for building web applications in Java. Servlets are Java classes that extend the functionality of web servers, allowing you to handle HTTP requests and generate HTTP responses. The javax.servlet package contains classes and interfaces that define the API for servlets and their interactions with the web container. Here are some of the key classes and interfaces within the javax.servlet package:

Servlet Interface:

javax.servlet.Servlet: This is the core interface that all servlets must implement. It defines the methods that servlets need to override, including init(), service(), and destroy(). Servlets process incoming requests and generate responses using the service() method.

ServletRequest and ServletResponse Interfaces:
javax.servlet.ServletRequest: This interface represents an incoming HTTP request and provides methods for accessing request parameters, headers, and other data.
javax.servlet.ServletResponse: This interface represents the HTTP response that will be sent to the client. It provides methods for setting response headers and writing content to the response.

HttpServletRequest and HttpServletResponse Interfaces:
javax.servlet.http.HttpServletRequest: This interface extends ServletRequest and provides additional methods specific to HTTP requests, such as methods for accessing HTTP headers, cookies, and session information.
javax.servlet.http.HttpServletResponse: This interface extends ServletResponse and provides additional methods for setting HTTP response status codes and managing cookies.

ServletConfig Interface:
javax.servlet.ServletConfig: This interface provides configuration information for a servlet, including initialization parameters specified in the web.xml file. It allows servlets to retrieve their initialization parameters.

ServletContext Interface:
javax.servlet.ServletContext: This interface represents the servlet context, which provides a way for servlets to interact with the servlet container and share information across multiple servlets. It includes methods for accessing resources and managing attributes.

GenericServlet Class:
javax.servlet.GenericServlet: This is an abstract class that implements the Servlet and ServletConfig interfaces. It can be extended to create servlets, simplifying some of the basic functionality.

HttpServlet Class:
javax.servlet.http.HttpServlet: This is an abstract class that extends GenericServlet and is designed for creating HTTP servlets. It simplifies handling HTTP-specific tasks like parsing parameters and managing sessions.

Lifecycle of a Servlet
There are three states of Servlet - new, ready, and end. The states of the servlets change with the help of the calling of servlets method.

The servlet interface provides a common functionality in all the Servlets. The servlet interface defines the methods that all the Servlets must implement. HttpServlet and GenericServlet implement servlet interfaces directly or indirectly. The servlet interface provides three lifecycle methods used to implement any Servlet.
A Serviet is loaded only once in the memory and is initialized in the init() method. After the Servlet is initialized, it starts accepting a request from the client and processes them through the service() method until it is shut down by the destroy() method. The service() method is executed for every incoming request. The lifecycle of a Servlet is depicted as follows:

1. Servlet class is loaded.
2. Servlet instance is created.
3. The init method is invoked.
4. The service method is invoked.
5. The destroy method is invoked

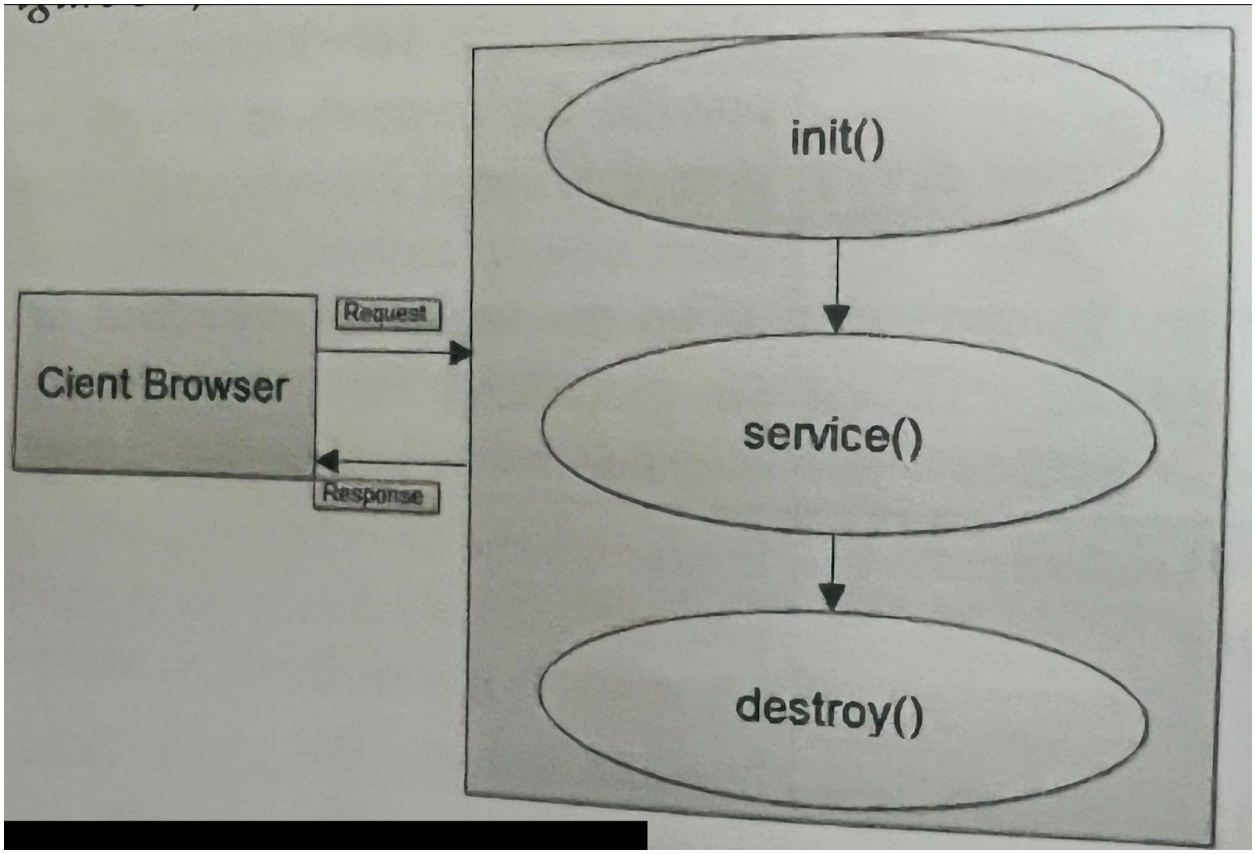
6. 

Table 3.3 describes the methods of the Servlet interface:

S.No	Method name	Description
1	`public void init(ServletConfig config) throws Servlet Exception`	It contains all initialization codes for the Servlet and is invoked when the Servlet is first loaded and created.
2	`public void service(ServletRequest request,ServletResponse response);`	It receives all the requests from clients, identifies the type of requests, and dispatches them to the `doGet()` or `doPost()` methods for processing.
3	`public void destroy()`	It executes once when the Servlet is removed from the server. The `cleanup` code for the Servlet must be provided in this method.
4	`public ServletConfig getServletConfig();`	It returns the object of `ServletConfig`.
5	`public String getServletInfo()`	It returns the information about the Servlet, such as writer, copyright, version, and so on.

Creating a Servlet

Along with three lifecycle methods(int(),service() and destroy()), two more methods are also used to create a Servlet.

Table 3.4 also describes two methods that are used in creating a Servlet:

S.No	Method name	Description
1	`ServletResponse.getWriter ()`	It returns a reference to a `PrintWriter` object. The `PrintWriter` class is used to write the formatted objects as a text-output stream onto the client.
2	`ServletResponse.setContentType (String type)`	It sets the type of content sent as a response to the client browser. For example, `SetContentType ("text/html")` is used to set the response type as text.

The Servlet can be created in three of the following ways:

- By implementing the Servlet interface
- By inheriting GenericServlet class
- By inheriting HttpServlet class

Servlet Interface Methods

Following are the methods of Servlet Interface:

Methods	Description
public void init(ServletConfig, config)	It is one of the Servlet life cycle methods. It is invoked by Servlet container after being initialized by Servlet.
public void service (ServletRequest req, ServletResponse res)	The **service( )** method is called after successful completion of **init( ).** It is invoked by Servlet container to respond to the requests coming from the client.
public ServletConfig getServletConfig( )	Returns a ServletConfig object, which contains initialization and startup parameters for this Servlet.
public String getServletInfo( )	Returns the information about the Servlet, such as author, version, and copyright. This method returns a string value.
public void destroy( )	Called by servlet container and it marks the end of the life cycle of a servlet. It indicates that servlet has been destroyed.

HttpServlet Class

The **HttpServlet** class extends the **GenericServlet** and implements the Serializable interface. It is an abstract class. The **HttpServlet** class reads the HTTP request from http, get, post, put, delete etc. It calls one of the corresponding methods.

HttpServlet Class Methods

- protected void doGet(HttpServletRequest req, HttpServletResponse resp)
- protected void doDelete(HttpServletRequest req, HttpServletResponse resp)

- protected void doHead(HttpServletRequest req, HttpServletResponse resp)
- protected void doPost(HttpServletRequest req, HttpServletResponse resp)
- protected void doPut(HttpServletRequest req, HttpServletResponse resp)
- protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
- protected void service(HttpServletRequest req, HttpServletResponse resp)
- public void (ServletRequest req, ServletResponse resp)

GenericServlet Class

It is an abstract class that implements **Servlet, ServletConfig** and **serializable** interface. It provides the implementation of all methods of these interfaces except the service method.

GenericServlet may be directly extended by Servlet. It provides simple versions of the life cycle methods **init( )** and **destroy( )** methods.

GenericServlet Class Methods

Following are the important methods of GenericServlet Class:

Methods	Description
public void destroy( )	Invoked by servlet container. It shows that the servlet is being taken out of service.
public String getInitParameter(String name)	Returns a String containing the value of named parameter.

public String getServletInfo( )	Returns information related to Servlet like author, version etc.
public String getServletName( )	Returns the name of Servlet object.
public void init( )	It is a convenience method that can easily be overridden so that we do not need to call **super.init(config).**
public void log(String msg)	Writes the given message to a Servlet log file.
public abstract void service(ServletRequest req, ServletResponse res)	It is an abstract method, called by the servlet container to allow the servlet to respond to a request.

Session Tracking/Session Management.

Introduction

- Session is basically a time frame and tracking means maintaining user data for certain period of time frame.
- **HTTP protocol** and **web servers** are stateless.
- All requests and responses are independent.
- Each request to the web server is treated as a new request.
- **Session Tracking** is a mechanism used by the web container to store session information for a particular user. It is used to recognize a particular user.

Methods of Session Tracking

There are four techniques used in Session Tracking:

1) Cookies

2) Hidden Form Field

3) URL Rewriting

4) HttpSession

1) Cookies

Cookies are small piece of information sent by web server in response header and gets stored in browser side. A web server can assign a unique session ID to each web client. The cookies are used maintain the session. The client can disable the cookies.

2) Hidden Form Field

The hidden form field is used to insert the information in the webpages and this information is sent to the server. These fields are not viewable to the user directly.

For example:

```
<input type = hidden'  name = 'session' value = '12345' >
```

3) URL Rewriting

Append some extra data through URL as request parameters with every request and response. URL rewriting is a better way to maintain session's management and work for the browsers.

For example:

```
http://localhost:8080/servlet.html?sessionid=54321
```

4) HttpSession Object

The HttpSession object represents a user session. The HttpSession interface creates a session between an HTTP client and HTTP server. A user session contains information about the user across multiple HTTP requests.

HttpSession

The HttpSession is a server-side mechanism for session tracking provided by the Java Servlet API. It allows developers to create and manage sessions on the server. When a user first accesses a servlet, the server creates a unique session and assigns it an ID. This session ID can be stored in a cookie or transmitted via other session tracking techniques. The HttpSession object on the server can store session-specific data, and it is accessible to multiple servlets within the same web application. This approach provides a robust and standardized way to handle session tracking in Java-based web applications.

Session Timeout

A session timeout specifies the amount of time a user's session can be inactive before it is automatically terminated. Inactivity refers to the user not interacting with the website in any way, like clicking links or submitting forms. Once the timeout period is reached, the server assumes the user has left or abandoned their session and clears out the associated data to free up resources.

In Servlets, you can configure session timeouts for a particular session using the setMaxInactiveInterval() method. This method takes a time duration in

seconds as an argument. For example, if you want to set a session timeout of 30 minutes, you can do this:

```
HttpSession session = request.getSession();
session.setMaxInactiveInterval(1800); // 1800 seconds = 30 minutes
```

Methods of HttpSession

Method	Description
setAttribute(String name, Object value)	Stores an attribute (key-value pair) in the session.
getAttribute(String name)	Retrieves the value of an attribute stored in the session, given its name.
removeAttribute(String name)	Removes an attribute from the session, given its name.
getId()	Returns a unique identifier for the session.

getCreationTime()	Returns the time when the session was created (in milliseconds since January 1, 1970, UTC).
getLastAccessedTime()	Returns the time when the session was last accessed (in milliseconds since January 1, 1970, UTC).
setMaxInactiveInterval(int interval)	Sets the maximum time interval, in seconds, between client requests before the session is invalidated due to inactivity.
getMaxInactiveInterval()	Retrieves the maximum inactive interval for the session in seconds.
invalidate()	Invalidates (destroys) the session, removing all session attributes and ending the session.
isNew()	Checks if the session is new (has just been created).

Implementation

To demonstrate the implementation of HttpSession methods in a Java servlet, I'll provide an example using a simple web application. In this example, we'll create a servlet that sets, retrieves, and invalidates session attributes. The output will be displayed on a web page.

Here's a step-by-step guide:

1. Create a new Java web project in your preferred Integrated Development Environment (IDE) such as Eclipse or IntelliJ IDEA.
2. Create a servlet class (SessionExampleServlet.java) in the project. This servlet will handle session-related operations.

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebServlet("/SessionExampleServlet")
public class SessionExampleServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        HttpSession session = request.getSession();
        String action = request.getParameter("action");
```

```
        if (action == null) {
            // Set session attribute
            session.setAttribute("message", "Hello, this is a
session attribute!");
            response.getWriter().println("Session attribute
set.");
        } else if (action.equals("get")) {
            // Get session attribute
            String message = (String)
session.getAttribute("message");
            response.getWriter().println("Session attribute
value: " + message);
        } else if (action.equals("invalidate")) {
            // Invalidate session
            session.invalidate();
            response.getWriter().println("Session
invalidated.");
        }
    }
}
```

3. Create a JSP page (index.jsp) to provide a simple HTML form for interacting with the servlet:

```
<!DOCTYPE html>
<html>
<head>
```

```
    <meta charset="UTF-8">
    <title>HttpSession Example</title>
</head>
<body>
    <h1>HttpSession Example</h1>
    <form action="SessionExampleServlet" method="GET">
        <input type="submit" name="action" value="Set Session
Attribute">
        <input type="submit" name="action" value="Get Session
Attribute">
        <input type="submit" name="action" value="Invalidate
Session">
    </form>
</body>
</html>
```

4. Configure the web.xml file or use servlet annotations, depending on your servlet container. For annotation-based configuration, ensure that the @WebServlet annotation in SessionExampleServlet.java specifies the correct URL mapping.
5. Deploy and run the web application on your servlet container (e.g., Apache Tomcat).
6. Access the application in a web browser by navigating to http://localhost:8080/your-web-app-context/index.jsp.
7. Interact with the web page to set, get, and invalidate the session attribute.
    - Click Set Session Attribute to set the session attribute.

- Click Get Session Attribute to retrieve and display the session attribute.
- Click Invalidate Session to invalidate the session.

How to Delete Session Data?

In Java web applications, you can delete session data by invalidating the session or by removing specific session attributes. Here's how to do it:

1. Invalidating the Session: To completely delete (invalidate) the entire session, including all associated session attributes, you can use the invalidate() method of the HttpSession object. Here's how to do it in a servlet:

```
HttpSession session = request.getSession();
session.invalidate();
```

When you call invalidate(), it immediately ends the session and removes all session data associated with it. Subsequent attempts to access the session or its attributes will create a new session.

2. Removing Specific Session Attributes: If you want to delete only specific session attributes while keeping the session active, you can use the removeAttribute(String name) method of the HttpSession object. Here's how to remove a specific session attribute:

```
HttpSession session = request.getSession();
session.removeAttribute("attributeName");
```

Replace attributeName with the name of the attribute you want to remove.

It's important to note that if you remove an attribute, it will no longer be available in the session. However, the session itself will remain active unless you explicitly call invalidate().

3. Combining Both Methods: If you want to both remove specific attributes and invalidate the session at the same time, you can do so in a servlet like this:

```
HttpSession session = request.getSession();
session.removeAttribute("attributeName");
session.invalidate();
```

This will remove the specified attribute and invalidate the session in one go.

For example:

HttpSession session = request.getSession( );

Session.setAttribute("username", "password");

Cookie

- **Cookies are small piece of data on the client computer that send response from the web server to client.**
- **They are used to store the client state.**

- **The information which is stored on the client machine is called cookie.**
- **A Servlet container sends small information to the web browser. This data is saved by the browser and later back to the server.**
- **The servlet sends cookies to the browser using HttpServletResponse.addCookie(javax.servlet.http.Cookie) method.**

Types of Cookies

Two types of cookies presents in Servlets:

1. Non-persistent/session cookie

2. Persistent cookie

1. Non-persistent/session cookie

Non-persistent cookies do not have an expiry time. They are valid for a single session only. The persistent cookies are live till the browser is open. They disappear when user closes the browser.

2. Persistent cookie

Persistent cookies have expiry time parameter. They are valid for multiple sessions. They do not disappear when user closes the browser. They are stored in primary memory of the computer. They disappear when user logs out or signs out.

Advantage of cookies:

1. The simplest technique for maintaining the state.

2. Cookies are stored on the client side.

Disadvantages of cookies:

1. It will not work if cookies are disabled in the browser.

2. Only textual information can be set in the Cookie object.

3. Cookie class

4. javax.servlet.http.Cookieclass provides the functionality of using cookies. Provides many useful methods for cookies.

Cookie Class Constructors

Constructor	Description
Cookie( )	**Constructs the cookie with default property.**
Cookie(String name, String value)	**Constructs a cookie with specified name and value.**

Cookie Class Methods

Following are some important methods of Cookie class:

Methods	Description
public String getName( )	**Returns the name of the cookies.**
public String getPath( )	**Returns the path of the server to which the browser returns the cookie.**
public String getValue( )	**Returns the value of the cookie.**
public int getMaxAge( )	**Returns the maximum age limit to the cookie, specified in seconds.**

public void setMaxAge(int expiry)	**Sets the maximum age of the cookies in seconds.**
public void setValue(String newValue)	**Allocates a new value to a cookie after the cookie is created.**

Create a Cookie Object

The constructor of Cookie class creates the cookie object with corresponding cookie name and value.

Example

```
Cookie cookie = new Cookie("username","Surendra");
Response.addCookie(cookie);
```

Reading Cookie Sent from the Browser

The getCookies( ) method is used for getting the cookie.

Example

```
Cookie[ ] cookies = request.getCookies( );
String username = null;
for (Cookie cookie : cookies)
{
   if("user".equals(cookie.getName( )))
   {
      username = cookie.getValue();
   }
}
```

Deleting the Cookies

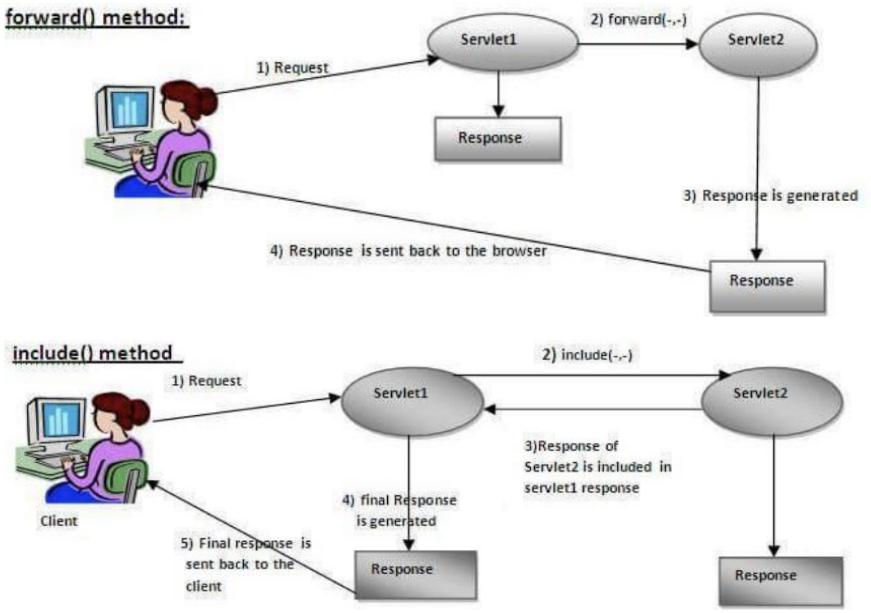
We can remove the cookies from the browser by setting the cookie expiry time to 0 or -1.

Example

```
Cookie cookie = new Cookie("user", "");
cookie.setMaxAge(0);
response.addCookie(cookie);
```

RequestDispatcher

- **The RequestDispatcher is an interface that defines an object to receive request from the client and sends them to any resource on the server.**
- **It implements an object to wrap together different types of resources in a Servlet container.**



RequestDispatcher Methods

The RequestDispatcher interface provides two methods:

Methods	Description
public void forward(ServletRequest request, ServletResponse response)	**It forwards a client request from a servlet to another resource (servlet, JSP file, HTML file) on the server.**
public void include(ServletRequest request, ServletResponse response)	**Includes the content of a resource (Servlet, JSP pages, HTML file) in the response.**

Getting the Object of RequestDispatcher

ServletRequest interface provides the getRequestDispatcher( ) method to returns the object of RequestDispatcher.

Example

```
RequestDispatcher rd = request.getRequestDispatcher("index.html");
rd.forward(request, response);
```

OR

```
RequestDispatcher rd = request.getRequestDispatcher("index.html");
rd.include(request, response);
```

Page Redirection

- **The Page redirection is the process of redirecting the response to another resource.**
- **It is used to move the document to the new location for load balancing or simple randomization.**
- **The sendRedirect( ) method is the method of HttpServletResponse interface.**
- **It is used to redirect response to another resource (servlet, jsp or HTML file).**

For example:

```
response.sendRedirect("index.html");
```

Difference between forward( ) method and sendRedirect( ) method

forward( )	sendRedirect( )
Used to forward the resources available within the server.	**Used to redirect the resources to different servers or domains.**
The forward( ) method is executed on the server side.	**The sentRedirect( ) method is executed on the client side.**
The forward( ) method is faster than sendRedirect( ).	**It is slower because each time a new request is created, old one is lost.**
The transfer of control is done by container and client/browser is not involved.	**The transfer of control is assigned to the browser and a new request to the given URL is initiated.**
The request is shared by the target resource.	**New request is created for the server resource.**
It is declared in RequestDispatcher interface.	**It is declared in HttpServletResponse.**

Example : Illustrating the sendRedirect( ) method for page redirecting in Servlet

```
//SendRedirectDemo.java


import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class SendRedirectDemo extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        response.sendRedirect("http://localhost:8080/")
        pw.close();
    }
}
```

Important Questions:

Q. What is the difference between a session and cookies?

A: A session is a server-side data storage mechanism, while cookies are small pieces of data stored on the client's browser. Sessions are often used to manage user state, whereas cookies are typically used for storing user-specific information on the client side.

Q. How do I set an expiration time for a session in Java servlets?

A: You can set the maximum inactive interval for a session in Java servlets using the setMaxInactiveInterval(int intervalInSeconds) method of the HttpSession object, specifying the desired time in seconds.

Q. Why is session management important in web applications?

A: Session management is crucial in web applications because it allows you to maintain user-specific data and interactions across multiple HTTP requests, enabling features like user authentication, personalized content, and shopping cart functionality.

Q. Can session data be shared between different web applications on the same server?

A: By default, session data is not shared between different web applications on the same server. Each web application has its own separate session management. However, you can configure your server to enable session sharing if needed, using mechanisms like cross-context session sharing in Java EE containers.

Java Server Pages (JSP)

Java Server Pages (JSP) is a programming tool on the application server side that supports platform-independent and dynamic methods to construct Web-based applications.

Much as Servlet technology does, the JSP method provides a web application. It can be considered an expansion of Servlet because it offers more features than servlet. Since we can differentiate design and development, the JSP pages are simpler to manage than Servlet. HTML tags and JSP tags are present in Java Server Pages.

To access enterprise servers, Java Server Pages has an approach to the entire community of Java APIs, including the JDBC API. This tutorial will walk you to the path of building your own web application in convenient and simple steps using Java Server Pages.

Why should we Learn JSP?

There are numerous reasons for us to learn JSP.

1) Its extension to Servlet technology will be the very first reason to learn JSP. In JSP, we can use all the functionality of Servlet. In addition, speech-language, predefined tags, implicit entities and custom tags can be used in JSP, making it easier for JSP to create.

2) The second reason would be that there is no need to redeploy and recompile the project in case the JSP page is modified. If we have to modify the look and sound of the programme, the Servlet code has to be revised and recompiled.

3) Third would be about how easy JSP is to maintain and manage as we can conveniently separate the presentation and business logic.

4) In JSP, we can use several tags which reduce the code, such as action tags, JSTL, custom tags, etc. We may, in addition, use EL, implied objects, etc.

Advantages of JSP

1) Extension to Servlet

2) Easy to maintain

3) Fast Development: No need to recompile and redeploy

4) Less code than Servlet

Life Cycle of a JSP Page

The JSP page follows these phases:

Translation of JSP Page

Compilation of JSP Page

Class loading (class file is loaded by the class loader)

Instantiation (Object of the Generated Servlet is created).

Initialization (jspInit() method is invoked by the container).

Request processing (_jspService() method is invoked by the container).

Destroy (jspDestroy() method is invoked by the container).

Note: jspInit(), _jspService() and jspDestroy() are the life cycle methods of JSP.

Directory Structure of JSP

The directory structure of JSP page is same as Servlet. We contains the JSP page outside the WEB-INF folder or in any directory.

Scripting Elements

The scripting element provides the ability to insert java code inside the JSP. There are three types of scripting elements:

Scriptlet tag

Expression tag

Declaration tag

JSP Scriptlet Tag

A Sriptlet tag is used to execute java source code in JSP. Syntax is as follows:

<% java source code %>

You can write XML equivalent of the above syntax as follows:

<jsp:scriptlet> code fragment </jsp:scriptlet>

Simple Example of JSP Scriptlet Tag

In this example, we are displaying a welcome message.

```
1. <html>
2. <body>
3. <% out.print("welcome to jsp"); %>
4. </body>
5. </html>
```

Example of JSP Scriptlet tag that prints the user name

index.jsp

```
<html>
<body>
<h2>this is index page</h2>
<jsp:forward page="printdate.jsp" />
</body>
</html>
```

printdate.jsp

```
<html>
<body>
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
</body>
</html>
```

Example of jsp:forward action tag with parameter

In this example, we are forwarding the request to the printdate.jsp file with parameter and printdate.jsp file prints the parameter value with date and time.

index.jsp

```
<html>
<body>
<h2>this is index page</h2>
<jsp:forward page="printdate.jsp" >
<jsp:param name="name" value="abc xyz" />
</jsp:forward>
</body>
</html>
```

printdate.jsp

```
<html>
<body>
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
<%= request.getParameter("name") %>
</body>
```

</html>

jsp:include action tag

The jsp:include action tag is used to include the content of another resource it may be jsp, html or servlet.

The jsp include action tag includes the resource at request time so it is better for dynamic pages because there might be changes in future.

Advantage of jsp:include action tag

code reusability

**Syntax of jsp:include
action tag without parameter**

**1.
<jsp:include page="relativeURL | <%= expression %>" />**

**Syntax of jsp:include
action tag with parameter**

```
<jsp:include page="relativeURL | <%= expression %>">
<jsp:param name="parametername" value="parametervalue |
<%=expression%>" />
</jsp:include>
```

Example of jsp:include action tag without parameter

In this example, index.jsp file includes the content of the printdate.jsp file.

File: index.jsp

```
<html>
<body>
<h2>this is index page</h2>
<jsp:include page="printdate.jsp" />
<h2>end section of index page</h2>
</body>
</html>
```

File: printdate.jsp

```
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
```

Setting Cookies with JSP

HTML File

```
<html>
<body>
<form action="main.jsp" method="GET">
First Name: <input type="text" name="first_name"><br />
Last Name: <input type="text" name="last_name" />
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

main.jsp

```
<% Cookie firstName = new Cookie("first_name",
request.getParameter("first_name"));

Cookie lastName = new Cookie("last_name",
request.getParameter("last_name"));

// Set expiry date after 24 Hrs for both the cookies.

firstName.setMaxAge(60*60*24);

lastName.setMaxAge(60*60*24);

response.addCookie( firstName );

response.addCookie( lastName );

%>

<html>

<body>

<center>
```

```
<h1>Setting Cookies</h1>

</center>

<ul>

<li><p><b>First Name:</b>

<%= request.getParameter("first_name")%>

</p></li>

<li><p><b>Last
Name:</b>

<%= request.getParameter("last_name")%>

</p></li>

</ul>

</body>

</html>
```

Reading Cookies with JSP

```
<html>

<body>

<center>

<h1>Reading Cookies</h1>

</center>

<%

Cookie cookie = null;

Cookie[] cookies = null;
```

```
// Get an array of Cookies associated with this domain

cookies = request.getCookies();

if( cookies != null ){

out.println("<h2> Found Cookies Name and Value</h2>");

for (int i = 0; i < cookies.length; i++){

cookie = cookies[i];

out.print("Name : " + cookie.getName( ) + ",
");

out.print("Value: " + cookie.getValue( )+" <br/>");

}

}else{
out.println("<h2>No cookies founds</h2>");
}

%>

</body>

</html>
```

Delete Cookies with JSP

To delete cookies is very simple. If you want to delete a cookie then you simply need to follow up following three steps:

Read an already existing cookie and store it in Cookie object.

Set cookie age as zero using setMaxAge() method to delete an existing cookie.

Add this cookie back into the response header.

Example:

```
<html>
<body>
<center>
<h1>Reading Cookies</h1>
</center>
<%
Cookie cookie = null;
Cookie[] cookies = null;
// Get an array of Cookies associated with this domain
cookies = request.getCookies();
if( cookies != null ){

out.println("<h2> Found Cookies Name and Value</h2>");

for (int i = 0; i < cookies.length; i++){
cookie = cookies[i];

if((cookie.getName( )).compareTo("first_name") == 0 ){

cookie.setMaxAge(0);

response.addCookie(cookie);

out.print("Deleted cookie: " +

cookie.getName( ) + "<br/>");

}

out.print("Name : " + cookie.getName( ) + ",
");

out.print("Value: " + cookie.getValue( )+" <br/>");

}

}else{
out.println(
"<h2>No cookies founds</h2>");
}
%>
</body>
</html>
```

HttpSession Object

PageCounter Example

```
<%@ page import="java.io.*,java.util.*" %>

<%

// Get session creation time.

Date createTime = new Date(session.getCreationTime());

// Get last access time of this web page.

Date lastAccessTime = new Date(session.getLastAccessedTime());

String title = "Welcome Back to my website";

Integer visitCount = new Integer(0);

String visitCountKey = new String("visitCount");

String userIDKey = new String("userID");

String userID = new String("ABCD");

// Check if this is new comer on your web page.

if (session.isNew()){

title = "Welcome to my website";

session.setAttribute(userIDKey, userID);

session.setAttribute(visitCountKey,
visitCount);

}

visitCount = (Integer)session.getAttribute(visitCountKey);

visitCount = visitCount + 1;
```

```
userID = (String)session.getAttribute(userIDKey);

session.setAttribute(visitCountKey,
visitCount);

%>

<html>

<head>
<title>Session Tracking</title>

</head>

<body>

<center>

<h1>Session Tracking</h1>

</center>

<table border="1" align="center">

<tr bgcolor="#949494">

<th>Session info</th>

<th>Value</th>

</tr>

<tr>

<td>id</td>

<td><% out.print( session.getId()); %></td>

</tr>

<tr>

<td>Creation Time</td>
```

```
<td><% out.print(createTime); %></td>

</tr>

<tr>

<td>Time of Last Access</td>

<td><% out.print(lastAccessTime); %></td>

</tr>

<tr>

<td>User ID</td>

<td><% out.print(userID); %></td>

</tr>

<tr>

<td>Number of visits</td>

<td><% out.print(visitCount); %></td>

</tr>

</table>

</body>

</html>
```

Database Operation in JSP

SELECT Operation

Following example shows how we can execute SQL SELECT statement using JTSL in JSP

programming:

```
<%@ page import="java.io.*,java.util.*,java.sql.*"%>

<%@ page import="javax.servlet.http.*,javax.servlet.*" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>

<head>
<title>SELECT Operation</title>

</head>

<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"

url="jdbc:mysql://localhost/TEST"

user="root"
password="pass123"/>

<sql:query dataSource="${snapshot}" var="result">

SELECT * from Employees;

</sql:query>

<table border="1" width="100%">

<tr>

<th>Emp ID</th>

<th>First Name</th>

<th>Last Name</th>

<th>Age</th>

</tr>
```

<c:forEach var="row" items="${result.rows}">

<tr>

<td><c:out value="${row.id}"/></td>

<td><c:out value="${row.first}"/></td>

<td><c:out value="${row.last}"/></td>

<td><c:out value="${row.age}"/></td>

</tr>

</c:forEach>

</table>

</body>

</html>

INSERT Operation

Following example shows how we can execute SQL INSERT statement using JTSL in JSP

programming:

<%@ page import="java.io.*,java.util.*,java.sql.*"%>

<%@ page import="javax.servlet.http.*,javax.servlet.*" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>

<head>

<title>JINSERT Operation</title>

</head>

```
<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost/TEST"

user="root"
password="pass123"/>

<sql:update dataSource="${snapshot}" var="result">

INSERT INTO Employees VALUES (104, 2, 'Nuha', 'Ali');

</sql:update>

<sql:query dataSource="${snapshot}" var="result">

SELECT * from Employees;

</sql:query>

<table border="1" width="100%">

<tr>

<th>Emp ID</th>

<th>First Name</th>

<th>Last Name</th>

<th>Age</th>

</tr>

<c:forEach var="row" items="${result.rows}">

<tr>

<td><c:out value="${row.id}"/></td>

<td><c:out value="${row.first}"/></td>

<td><c:out value="${row.last}"/></td>
```

<td><c:out value="${row.age}"/></td>

</tr>

</c:forEach>

</table>

</body>

</html>

DELETE Operation

Following example shows how we can execute SQL DELETE statement using JTSL in JSP

programming:

<%@ page import="java.io.*,java.util.*,java.sql.*"%>

<%@ page import="javax.servlet.http.*,javax.servlet.*" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>

<head>

<title>DELETE Operation</title>

</head>

<body>

<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost/TEST"

**user="root"
password="pass123"/>**

```
<c:set var="empId" value="103"/>
<sql:update dataSource="${snapshot}" var="count">
DELETE FROM Employees WHERE Id = ?
<sql:param value="${empId}" />
</sql:update>
<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>
<table border="1" width="100%">
<tr>
<th>Emp ID</th>
<th>First Name</th>
<th>Last Name</th>
<th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
<td><c:out value="${row.id}"/></td>
<td><c:out value="${row.first}"/></td>
<td><c:out value="${row.last}"/></td>
<td><c:out value="${row.age}"/></td>
</tr>
```

</c:forEach>

</table>

</body>

</html>

UPDATE Operation

Following example shows how we can execute SQL UPDATE statement using JTSL in JSP

programming:

<%@ page import="java.io.*,java.util.*,java.sql.*"%>

<%@ page import="javax.servlet.http.*,javax.servlet.*" %>

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<html>

<head>

<title>DELETE Operation</title>

</head>

<body>
<sql:setDataSource var="snapshot" driver="com.mysql.jdbc.Driver"

url="jdbc:mysql://localhost/TEST"

user="root"
password="pass123"/>

<c:set var="empId" value="102"/>

<sql:update dataSource="${snapshot}" var="count">

```
UPDATE Employees SET last = 'Ali'
<sql:param value="${empId}" />
</sql:update>
<sql:query dataSource="${snapshot}" var="result">
SELECT * from Employees;
</sql:query>
<table border="1" width="100%">
<tr>
<th>Emp ID</th>
<th>First Name</th>
<th>Last Name</th>
<th>Age</th>
</tr>
<c:forEach var="row" items="${result.rows}">
<tr>
<td><c:out value="${row.id}"/></td>
<td><c:out value="${row.first}"/></td>
<td><c:out value="${row.last}"/></td>
<td><c:out value="${row.age}"/></td>
</tr>
</c:forEach>
</table>
```

```
</body>

</html>
```